Interdependence of Methods and Representations in Design of Software for Combinatorial Optimization

Collette Coullard

Department of Industrial Engineering and Management Sciences
Northwestern University
2225 North Campus Drive
Evanston, IL 60208-3119, U.S.A.

708-491-3077 (voice)
708-491-8005 (fax)

coullard@iems.nwu.edu

Robert Fourer

Department of Industrial Engineering and Management Sciences
Northwestern University
2225 North Campus Drive
Evanston, IL 60208-3119, U.S.A.

708-491-3151 (voice)
708-491-8005 (fax)

4er@iems.nwu.edu

Technical Report 95-67,
Industrial Engineering and Management Sciences, Northwestern University.

Interdependence of Methods and Representations in Design of Software for Combinatorial Optimization

Abstract. Practical algorithmic *methods* for combinatorial optimization problems cannot be considered in isolation from the *representations* that people use when communicating these problems to computer systems. Different representations define different problem classes, for which distinct types of methods are appropriate. Conversely, different methods have different ranges of applicability, which have motivated a diverse variety of representations.

This strong interdependence of method and representation in combinatorial optimization is quite the opposite of what one finds in linear or continuous nonlinear programming, where a single standard form permits communication between numerous methods and representations that have been independently developed. The most evident consequence of this interdependence has been a tendency for different research groups to address combinatorial optimization in separate ways, through different preferred combinations of representations and methods. A related consequence has been a lack of good *general purpose* combinatorial optimization compter packages.

We address in this paper the possibility that contrasting representations and their associated methods can in fact be brought together, to considerable advantage. To this end, we survey three representations popularly applied in combinatorial optimization: algebraic modeling languages, constraint logic programming languages, and netforms (or network diagrams). We first describe the kinds of optimization methods and systems most commonly associated with these alternatives. Each pair of representations is then considered, to show how each has been advantageous and how its advantages have begun to influence (or ought to influence) the design of the other. Our current research projects are described in conjunction with several of these comparisons.

Practical algorithmic *methods* for combinatorial optimization problems cannot be considered in isolation from the *representations* that people use when communicating these problems to computer systems. Different representations define different problem classes, for which distinct types of methods are appropriate. Conversely, different methods have different ranges of applicability, which have motivated a diverse variety of representations.

This strong interdependence of method and representation in combinatorial optimization is quite the opposite of what one finds in linear or continuous nonlinear programming, where a single standard form permits communication between numerous methods and representations that have been independently developed. One consequence has been the tendency of separate research groups — such as the “AI” and “OR” communities — to address optimization in separate ways, through different preferred combinations of representations and methods. Another consequence is a lack of good general purpose combinatorial optimization packages, as one can see by examining the many ads for other optimization systems in a typical issue of *OR/MS Today*.

Which kinds of representation for combinatorial optimization are most deserving of study? In principle, any sufficiently rich programming environment can be regarded as a general and powerful modeling system. The programming language may be entirely general (C, Fortran), or specialized for mathematical modeling (Matlab, Mathematica, Maple), or specialized for optimization as in the case of decades-old packages such as OMNI [29] and modern C++ libraries such as ILOG Solver [53]. The drawbacks of programming to describe optimization models are well known, however; so-called "matrix generation" programs are hard to debug, to maintain, and to document [15].

Our focus in this paper is on higher-level representations that allow optimization models to be described non-procedurally, in terms familiar to human modelers. The continuing success of such representations — and of modeling systems based on them — testifies to their value in optimization. We survey in particular three representations popularly applied in combinatorial optimization,

- algebraic modeling languages,
- constraint logic programming languages, and
- netforms, or network diagrams.

We first describe the kinds of optimization methods and systems most commonly associated with these alternatives. Each pair of representations is then considered, to show how each has been advantageous and how its advantages have begun to influence (or ought to influence) the design of the other. Our current research projects are described in conjunction with several of these comparisons.

1. Algebraic modeling languages

We take it for granted that virtually any problem of optimizing a linear function of given decision variables, subject to linear equations and inequalities in the variables, can be solved by any of several general-purpose algorithms. This remarkable property has permitted the development of comprehensive *modeling languages* for the support of linear programming.

The idea of a modeling language is to describe a linear program in a form that is natural for people to work with, yet that can be translated by a computer system to forms that are required by optimizing algorithms. Subscripted data and variables are concisely and symbolically defined by the language as collections of components indexed over sets; explicit data values are provided separately in text or or database tables. The objective and constraint equations and inequalities may be specified through the use of algebraic expressions (AIMMS [5], AMPL [17, 18], GAMS [8, 9], LINGO [52], MGG [55], MPL [44], SML [21, 22]), through a description of the activities associated with variables (AMPL again, MathPro [30]) or through a description of the block structure of the constraint matrix (MIMI [10], PAM [63]).

Our particular concern in this paper is with *algebraic* modeling languages, which have exhibited the greatest potential for extension beyond linear programming. Algebraic languages are based on familiar mathematical terminology for functions and comparisons, but with modifications to use the ASCII character set and to permit unambiguous interpretation by a computer system. Thus in AMPL — whose design is one current focus of our research — one may for example write

$$\text{minimize } \textstyle\sum_{i\in I}\sum_{j\in J}\sum_{t=1}^{T} c_{ijt}x_{ijt}$$

as

```
minimize Total_Cost:
   sum {i in I} sum {j in P} sum {t in 1..T} c[j,t] x[j,t];
```

Similarly, an algebraic constraint such as

$$\sum_{i \in I} l_{ij} x_{ijt} + z_{j,t-1} - z_{jt} = d_{jt}, \quad \text{for all } j \in J,\ t = 1, \ldots, T$$

may transcribed to AMPL as

```
subj to Demand {j in J, t = 1..T}:
   sum {i in I} l[i,j] * x[i,j,t] + z[j,t-1] - z[j,t] = d[j,t];
```

The transcription is shown here in its most literal form, to emphasize similarity to the mathematical statement. In practice, the AMPL model's components are often given longer, more meaningful names:

```
subject to Demand {j in DEST, t in 1..nweeks}:
   sum {i in ORIG} loss[i,j] * Ship[i,j,t]
      + Inv[j,t-1] - Inv[j,t] = demand[j,t];
```

The analogy to the original mathematical statement remains strong nevertheless. Sets such as `DEST` and `ORIG`, numerical parameters such as `nweeks` and `demand`, and variables such as `Ship` and `Inv` are declared by separate AMPL statements that precede their use in the objective and constraints.

By admitting variables declared as integer-valued, an algebraic modeling language may specify a broad range of combinatorial optimization problems. Integer variables are most natural when they represent quantities that are physically meaningful, but that cannot take fractional values in an implementable solution. In Figure 1–1, for example, the variables of a cutting-stock model must be integral because it is impractical to cut a fraction of a roll. Figure 1–2a shows a "mixed-integer" model of production and distribution, in which the factories are required to use only whole numbers of crews (as represented by the variables `Work[f]`). The variables

```
param nPAT integer >= 0, default 0;
param roll_width;

set PATTERNS := 1..nPAT;
set WIDTHS;

param orders {WIDTHS} > 0;
param nbr {WIDTHS,PATTERNS} integer >= 0;

   check {j in PATTERNS}: sum i in WIDTHS i * nbr[i,j] <= roll_width;

var Cut {PATTERNS} integer >= 0;

minimize Number: sum {j in PATTERNS} Cut[j];

subj to Fill {i in WIDTHS}:
   sum {j in PATTERNS} nbr[i,j] * Cut[j] >= orders[i];
```

Figure 1–1. *A simple cutting-stock model in AMPL [17, 18].*

```
var Work {f in fact} integer >= cr_min[f], <= cr_max[f];

var Manu {p in prd, f in fact: pc[p,f] <> 0} >= 0;

var Ship {p in prd, (d,w) in ship_rt} >= 0;

# --------------------------------------

minimize cost:

   sum {p in prd, f in fact: pc[p,f] <> 0}
      (pc[p,f]*dp[f]*hd[f]/pt[p,f]) * Manu[p,f] +

   sum {p in prd, (d,w) in ship_rt}
      (sc[d,w]*wt[p]*cpp[p]/1000) * Ship[p,d,w];

# --------------------------------------

subj to CREWS:
   cr_total_min <= sum {f in fact} Work[f] <= cr_total_max;

subj to P {f in fact}:
   Work[f] = sum {p in prd: pc[p,f] <> 0} Manu[p,f];

subj to D {p in prd, d in fact}:
   dp[d]*hd[d]/(pt[p,d]*cpp[p]/1000) * Manu[p,d]
      >= sum {(d,w) in ship_rt} Ship[p,d,w];

subj to W {p in prd, w in whse}:
   sum {(d,w) in ship_rt} Ship[p,d,w] = dem[p,w];
```

Figure 1–2a. *Part of an AMPL production-distribution model.*

```
var Ship_Use {(d,w) in ship_rt: d <> w} binary;

subject to Ship_Min {(d,w) in ship_rt: d <> w}:

   sum {p in prd} Ship[p,d,w]
      >= min (dsr[d], sum {p in prd} dem[p,w]) * Ship_Use[d,w];

subject to Ship_Max {(d,w) in ship_rt: d <> w}:

   sum {p in prd} Ship[p,d,w]
      <= (sum {p in prd} dem[p,w]) * Ship_Use[d,w];
```

Figure 1–2b. *Algebraic constraints to require that positive shipments be greater than a specified threshold.*

for amounts manufactured and amounts shipped are allowed to be fractional, however, because any fractional parts are so small relative to the total output that they may be rounded without any practical effect on the feasibility or optimality of the solution.

The collection of integer variables in Figure 1–2b (denoted `Ship_Use[d,w]`) is of a different nature. These are binary variables, restricted to the values 0 and 1, which have a logical rather than a physical interpretation. The constraints `Ship_Min` and `Ship_Max` are cleverly constructed so that total shipments from `d` to `w` are fixed at zero if `Ship_Use[d,w]` = 0, but are bounded below by some positive value if `Ship_Use[d,w]` = 1. Constraints of this kind do not correspond directly to the modeler's simple original conception that if shipments are positive along any link, they must be at least a certain minimum. As a result, an underlying design principle of the modeling language — that the language should express constraints in much the same way that the modeler conceives them — is violated. When these constraints are added to those of Figure 1–2a, however, the result is a useful model in which the offending constraints are just a small part.

To solve linear programs that have integer variables, algebraic modeling language systems employ "branch-and-bound" solvers that are built on top of ordinary continuous linear programming codes. The linear programming features of these systems, such as their routines for sending problems to solvers and for displaying results, thus carry over directly for integer programming.

Branch-and-bound solvers make use of an LP code by solving continuous relaxations of subproblems that arise in the search for the optimum. The relaxations provide lower bounds on the optimum (of a minimization) that, combined with the upper bounds provided by feasible integer solutions, allow the search tree to be pruned to a tractable size. The branch-and-bound approach works best when the relaxations provide relatively tight lower bounds, and tight bounds are in turn most likely when the relaxed variables are physical rather than logical in nature. Thus the integer problems that are most naturally formulated using algebraic modeling languages also tend to be the ones most readily addressed by the solvers used with these languages.

The best branch-and-bound solvers do often succeed on integer programs that contain logical variables, thanks to a variety of extensions and enhancements. We will return to these below in the comparison with constraint logic programming.

2. Constraint logic programming languages

Languages designed for the purpose of describing logic problems can be extended in a natural way to encompass combinatorial optimization. Lauriere's ALICE [45], the most notable early work in this area, describes an optimization problem in terms of finding a best function of a certain kind; constraints are described through a variety of algebraic and logical forms. The work of Van Hentenryck and others in the context of the CHIP project [14, 60, 61], has more recently attracted attention. CHIP is based on the Prolog language [58], an established tool for logic programming in artificial intelligence; other Prolog-based systems include CLP(R) [34], ECLiPSe [1] and Prolog III [11]. An alternative line of development has employed something resembling a conventional programming language, but with procedures designed so that they can be used in a declarative way. This approach can use custom-developed languages, but can also work well with general-purpose object-oriented languages such as C++; examples include 2LP [47], CHARME, ILOG Solver [53] and Oz [54].

An interesting example is provided by Jordan and Drexl [40], who compare the efficiency of a constraint logic approach and an integer programming approach for a batch sequencing problem. Their "conceptual model formulation" has the objective

$$\text{minimize } \sum_{j=1}^{J} e_j(d_j - X_j) + SC_{S_{j-1}S_j}$$

where d_j, e_j are the deadline and "earliness penalty" for job j, SC_{ij} is the setup cost between job i and job j, and variable X_j is the completion time of job j. The sequencing constraints are

```
schedule(R,S,X) :-

  R = [R1,R2], R :: 1..2,
  S = [S1,S2], S :: 1..2,
  X = [X1,X2], X :: 0..33,

  min_max(constrain(R,S,X,SC_S0S1,SC_S1S2),
          9*(21 - X1) + 6*(33 - X2) + SC_S0S1 + SC_S1S2 ).

constrain(R,S,X,SC_S0S1,SC_S1S2) :-

  R = [R1,R2],
  alldistinct(R),
  labeling(R),

  S = [S1,S2],
  alldistinct(S),
  labeling(S),

  element(R1,[S1,S2],1),
  element(R2,[S1,S2],2),

  element(S1,[6,3],P_S1),
  element(S2,[6,3],P_S2),

  element(S1,[3,2],ST_S0S1),
  element(S1,[[0,1],[3,0]],STRow_S1),
  element(S2,STRow_S1,ST_S1S2),

  X = [X1,X2],
  labeling(X),

  X1 #<= 21,
  X2 #<= 33,

  element(S1,[X1,X2],X_S1),
  element(S2,[X1,X2],X_S2),

  X_S1 #>=  ST_S0S1 + P_S1,
  X_S2 #>=  X_S1 + ST_S1S2 + P_S2,

  element(S1,[90,60],SC_S0S1),
  element(S1,[[0,30],[90,0]],SCRow_S1),
  element(S2,SCRow_S1,SC_S1S2),

  Cost #= 9*(21 - X1) + 6*(33 - X2) + SC_S0S1 + SC_S1S2.
```

Figure 2–1. *The batch sequencing problem in ECLiPSe, a Prolog-based system, for a simple two-job instance.*

$$\begin{array}{ll}
X_j \leq d_j, & \text{for all } j = 1, \ldots, J \\
X_{S_{k-1}} + ST_{S_{k-1}S_k} + p_{S_k} \leq X_{S_k}, & \text{for all } k = 1, \ldots, J \\
R_i < R_j, & \text{for all } i < j \text{ where } i, j \in M_m,\ m = 1, \ldots, N \\
S_{R_j} = j, & \text{for all } j = 1, \ldots, J \\
X_0 = 0, \quad S_0 = 0 &
\end{array}$$

where d_j, p_j are the deadline and processing time for job j, ST_{ij} is the setup time between job i and job j, and M_m is a subset of jobs belonging to job class $m = 1, \ldots, N$. In addition to X_j as before, the variables S_k, R_j are the job in position

```
define class(P,
               X,S)
{
array Pos :: [1..JJ];

for I in 0..(IT-1) do
   for K in (I*JpI + 1)..((I+1)*JpI) do {
 K = S[Pos[K]];
 }

for I in 0..(IT-1) do
   for K in (I*JpI + 2)..((I+1)*JpI) do {
   Pos[K] < Pos[K-1];
   X[K] <=  X[K-1] - P[K-1];
  }
}

define early(D,
               X)
{
for J in 1..JJ do {
   X[J]<= D[J];
   }
}

define sequence(P,ST,
     X,S)
{
all_diff(S);
for K in 1..JJ do  {
   X[S[K]] >= X[S[K-1]] + P[S[K]] + ST[S[K-1],S[K]];
   }
}

define cost(E,SC,D,
           X,S,
               Costs)
{
array C_of_J::[1..JJ];
for K in 1..JJ do  {
    C_of_J[K] = E[K]*(D[K] - X[K]) + SC[S[K-1],S[K]];
    }
sum(C_of_J,Costs);
}
```

Figure 2–2. *Constraints and objective for the batch sequencing problem in CHARME, a C-like language. This is part of a CHARME program used by the authors of [40] in comparisons of constraint logic programming and integer programming solvers.*

k and the position of job j, respectively. These variables represent permutations of $1, \ldots, J$, and appear in the subscripts of the key precedence constraint; hence this is not a conventional algebraic formulation.

Figure 2–1 exhibits a representation of this problem in a Prolog-based language (ECLiPSe), for a small instance, while Figure 2–2 presents an excerpt from the more general representation in a C-like language (CHARME) that was used for the experiments in [40]. The key relation of the problem statement in either figure to the above conceptual formulation is not that it has an especially similar appearance, but that it uses the same decision variables and imposes the same constraints. By contrast, the algebraic integer programming approach described in §1 requires a reformulation in terms of additional zero-one variables as shown in [40].

To perform optimization, constraint logic programming systems employ a variety of general-purpose methods for searching the solution space. Reflecting the Prolog heritage of these systems, some of their search methods are descended from the standard (depth-first) backtracking procedures that have long been employed in Prolog systems. Efficient optimization normally requires more sophisticated search strategies, however, that can backtrack more "intelligently" and that can apply forward-checking procedures to eliminate large parts of the solution space before they are reached by the search; often these strategies can be controlled by the modeler or can take advantage of problem-specific information that the modeler supplies. Some systems can also generate and use bounds as described in §1, but they do not necessarily rely on anything analogous to the "LP relaxations" solved for lower bounds in integer programming codes.

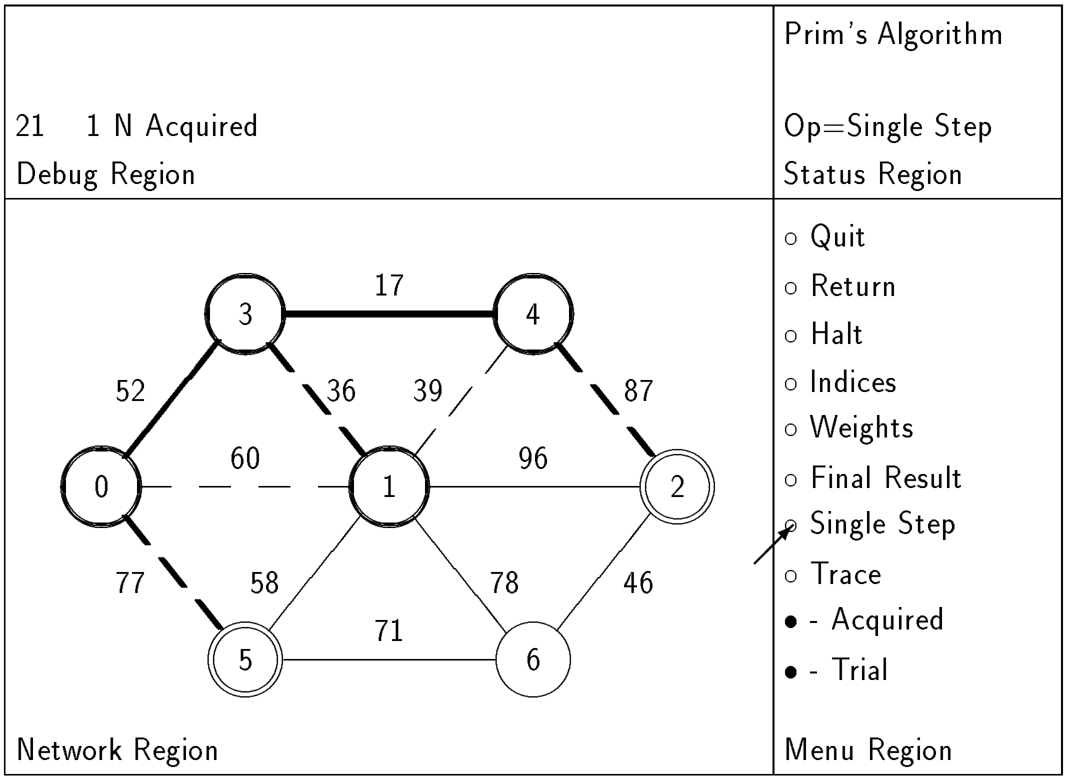
3. Network-based representations

One of the largest and best-known classes of combinatorial optimization problems are those that can be posed in terms of networks: nodes, arcs connecting nodes, and data associated with arcs and nodes. Glover, Klingman and Phillips have formalized this representation, introducing the term *netform* to denote a general network data structure together with conventions for representing instances of the structure as network diagrams [25, 26]. Steiger, Sharda and Leclaire's GIN [56, 57] implements netforms within a model-management system for minimum-cost network flow problems; other examples include Ogryczak, Studziński and Zorychta's DINAS/EDINET [50, 51], McBride's NETSYS [48], Jones's NETWORKS [35, 36, 37, 38], and Kendrick's PTS [43].

Figure 3–1 presents a representative view of a simple network as displayed by GIDEN, a network-based optimization system that is one focus of our current research. This view comes from an intermediate stage of Prim's algorithm for finding a minimum spanning tree. It is seen only when the "single stepping" mode is on; otherwise the user sees only the original network (all thin arcs) and the final one (spanning tree arcs changed to thick). The network representations generated by this kind of system are notable for incorporating no actual indication that any one kind of network problem is to be solved. Faced with the network from Figure 3–1, we could just as well have asked GIDEN to find, say, shortest paths by Dijkstra's algorithm (in which case it would have prompted us to designate the starting node).

Network-based optimization systems are thus properly regarded as "toolbox" systems that provide the user with one general netform representation together with a library of models based on that representation. In this respect they resemble typical statistical systems, with netforms being the analogue of data series. In contrast to the user of an algebraic modeling language system or constraint logic programming system, the user of a toolbox system does not define new models, but rather works with models already defined by the system's developer. The system is consequently very easy to use in contexts where its models apply, and its interface can be tailored to manipulations of the system's data structure. Such a system tends to have hard limits to what it can do, however, as defined by the available models and possibly combinations of them.

A toolbox system necessarily requires a collection of algorithms to implement its models. Netform toolbox systems are thus well positioned to take advantage of the bulk of research in network optimization, which has produced hundreds of provably efficient, narrowly targeted algorithms. (The state of the art is well described for network flow problems by Ahuja, Magnanti, and Orlin [2], and for a variety of other network-based problems by Cook, Cunningham, Pulleyblank, and Schrijver [12].) The toolbox approach is also useful for collecting varied heuristics that have been devised for well-studied NP-complete problems such as the traveling salesman problem.


21 1 N Acquired

Debug Region

Prim's Algorithm

Op=Single Step

Status Region

Figure 3–1. *Sample network display generated by GIDEN, while applying Prim's method to find a minimum spanning tree.*

4. Algebraic vs. constraint languages

Algebraic modeling languages can be contrasted to constraint logic programming languages in many respects. We organize the following presentation by first comparing the systems' underlying motivation, and then their system architectures, languages, and algorithms. A final sub-section discusses likely future interrelationships between these two kinds of systems, especially where one is likely to benefit by taking some ideas from the other.

Motivation. All designs of modeling systems for optimization are ultimately driven by empirical considerations: the needs of users to solve certain problem classes, and the desire of users to work in terms of familiar concepts.

In the design of constraint logic programming systems, empirical concerns are balanced against a solid grounding in the theory of logic programming. This theory has a substantial history of study in computer science, and continues to influence system design. It has not been unusual for new languages to arise in the context of research projects.

In contrast, algebraic modeling languages are based almost entirely on empirical concerns, especially the principle that the language's representation should correspond to the modeler's conception as closely as possible. There has been correspondingly little foundation in theory. Some interesting prototypes have been developed in schools of business and departments of industrial engineering and operations research, notably around the theory of "structured modeling" expounded by Geoffrion [19, 20]. But today's most widely used systems are either entirely commercial in origin, or have their background in non-commercial projects (AMPL [18], GAMS [9]) that emphasized user support and that eventually spun off marketable products.

System architecture. Constraint logic programming systems are almost exclusively implemented as extensions to logic programming systems. Extensions to Prolog were most common initially, but even in other cases the constraint features tend to be designed as extensions to a logic programming base. Such an approach has the advantage of facilitating development and updating, but the disadvantage of offering less flexibility in the design of the optimization component.

Each popular algebraic modeling language, by contrast, is designed as the focus for a self-contained optimization system. These systems are consequently somewhat difficult to develop and modify, with the result being a slow rate of change and a strong position for languages that have built up a user base. On the other hand, because there are no requirements being imposed by the design of an embedding language (such as Prolog), this approach promotes the design of modeling languages that are especially natural and convenient for optimization users. (There has been some experimentation with the idea of a linear programming language embedded in C++ [49], but this remains for now at an experimental stage.)

Every constraint logic programming system offers its own procedures for searching the solution space, together with language features that help the user to refine the search strategy. These features are often a main selling point when comparing one system to another. It is hard to compare the search routines from two different developers, however, except by formulating models in two different languages.

Algebraic modeling languages are generally implemented independently of optimization software. A language may offer connections to a dozen or more solvers, providing an excellent environment for comparison of optimization products. Conversely, the market for solvers is independent of the market for languages, with major solver vendors CPLEX [13] and OSL [33] offering links to variety of different languages. This environment has tended to encourage both language and solver development, as the developer of a new language can target essentially all solvers, and vice-versa. The interface between languages and solvers does tend to be fairly general purpose in design, however, with the result that users may not be taking advantage of specialized solver features as much as they should.

Languages. Algebraic modeling languages tend to be most appropriate to problems that have a natural representation in terms of arithmetic expressions in decision variables, while constraint logic programming languages have the edge for problems most naturally represented in terms of logical predicates. This is hardly surprising; indeed, the design principles of the algebraic languages suggest that they should only be used for problems that people conceive in algebraic terms. There is room for disagreement as to how people conceive (or should conceive) even simple linear programs, however — see for example [41, 63] — and so we should expect no sharp dividing line between the problems best expressed by the algebraic and by the constraint logic languages.

As an example, the algebraic representation of the cutting stock problem in Figure 1–1 would strike many people as very natural, since the variables correspond directly to the decisions that must be made: how many rolls to cut according to each pattern. Yet the cutting stock problem is also a popular example for constraint logic programming [60, pp. 181–187]. The logical aspect of the problem becomes more pronounced when there are *ad hoc* operational constraints, and when the generation of the patterns is complicated by problem-specific rules or by cutting in more than one dimension.

Algorithms. For either of the systems we are considering here, the commonly used optimization methods amount to sophisticated forms of tree search. Conventionally, these methods have differed most clearly in how they prune the tree: by purely logic-based lookahead methods for constraint logic programming algorithms, or by use of LP relaxations for integer programming codes. Heuristics for setting node and branch search priorities have differed accordingly, as in the case of the "pseudocost" criteria that depend on the dual values from the LP relaxation. Since all search-based optimizers are being applied to NP-complete problems, however, no one search strategy can be expected to be most efficient for more than a fraction of the problems that people want to solve. Successful implementations must depend on the accumulation of a variety of useful options. Through this process, tree-search routines have tended to adopt somewhat the same ideas, whatever their origins.

Branch-and-bound codes for integer programming have long handled a few kinds of logical constraints specially to enhance performance. Most common is the constraint that, from a given collection of nonnegative variables, at most one may be positive. This goes by the name of a "special ordered set of type 1" (or "SOS1"), and is generally accompanied by some facility for influencing the branching priority among members of the set. If this restriction were instead to be written entirely in terms of algebraic constraints, then in general an additional binary variable would have to be declared corresponding to each member of the special ordered set. The AIMMS language [5] has a way of declaring that the variables appearing in a particular constraint are also the members of a SOS1, but as yet no fully general format for SOS1 constraints has been introduced into an algebraic modeling language.[†]

A special ordered set of type 2 similarly restricts at most two adjacent variables of the set to be positive, and is intended for piecewise-linear terms in individual variables. In the case of AMPL, which has its own algebraic notation for piecewise-linear functions, each piecewise-linear term is translated to use one of these "SOS2" constraints (except in special cases where transformation to purely linear constraints is sufficient).

Conventional branch-and-bound codes can also be extended to handle the special case in which a variable must be either zero or $\geq$ some positive lower bound [28]. This facility is offered by the XA mixed-integer programming solver, but does not yet seem to be supported by any algebraic modeling language.

From the opposite standpoint of constraint logic programming, solvers routinely handle much more general forms of logical conditions, since they have their origins in solving logic programs. When extended to numerical variable types and to optimization, however, they vary considerably in features provided for the tree search. Some can generate bounds automatically, for example, while others rely on bound expressions provided by the user. There does not seem to be much standardization of extensions (in the way that special ordered sets have become standard for LP-based branch-and-bound).

Future interrelationships. The two kinds of language considered in this section embrace fundamentally different approaches to describing optimization problems. Yet each has a history of adopting some of the other's features to increase its expressive power. Constraint logic programming languages arose from logic programming languages such as Prolog by adopting a broader variety of domains for variables and more general algebraic predicates. Algebraic modeling languages have attempted to represent some of the logical constraints represented by special ordered sets. Further progress is likely to depend less on advances in language design, however, than on the coordination of language extensions with algorithmic advances, as we now proceed to explain with a few examples.

The expressiveness of algebraic modeling languages could readily be extended in a number of ways by adding logical functions and operators and by extending the contexts in which they may be used. By merely allowing such operators as `or` and `if ... then ... else` to be applied more generally to variables, algebraic languages can naturally express a great variety of combinatorial constraints that would otherwise require the introduction of formulation tricks involving zero-one variables. The awkward constraints previously noted in Figure 1–2b could be replaced, for example, by

[†]The only notable exception is where all variables in the special ordered set are already binary. This case and a few generalizations, sometimes called special ordered sets of type 3, are easy to detect automatically. Thus for instance the AMPL/CPLEX 3.0 user can write a SOS3 constraint as an equivalent algebraic AMPL constraint in binary variables, after which CPLEX's preprocessor can detect all such constraints and convert them back to their original logical forms for the branch-and-bound procedure.

```
subject to Ship_Size {(d,w) in ship_rt: d <> w}:
   sum {p in prd} Ship[p,d,w] = 0 or
   sum {p in prd} Ship[p,d,w]
      >= min (dsr[d], sum {p in prd} dem[p,w]);
```

AMPL's `card` function (cardinality, or number of members in a set) could also be allowed to apply to variables; also the iterated forms of the `or` and the `and` operator, `exists` and `forall`, could be supplemented by such operators as `atmost` and `exactly`. Then we could write such constraints as

```
subject to Ship_Exclusive {p in prd, w in whse}:
   card {(d,w) in ship_rt: Ship[p,d,w] > 0} <= 1;
```

or

```
subject to Ship_Exclusive {p in prd, w in whse}:
   atmost(1) {(d,w) in ship_rt} Ship[p,d,w] > 0;
```

These are readily seen to be equivalent ways of specifying the logical intent of a special ordered set of type 1.

The major obstacle to such extensions has been the absence of any branch-and-bound implementations that can deal with the broad range resulting logical expressions. Ideas from constraint logic programming methods, which are grounded in the broadly applicable theory of logic programming, may help to overcome this difficulty, although the difference in system architectures discussed above is likely to pose a problem, however. The open-interface design of algebraic modeling systems might enable a constraint logic programming solver to be hooked in as just one more alternative, except that the tightly integrated design of constraint logic systems does not normally provide for separating out the solver as a standalone system. Progress in this area may thus have to await the development of new solvers. There is also the possibility of further logic-based extensions to current branch-and-bound solvers; the growing popularity of "supernode" processing [33, 59] is a step of a sort in this direction.

The expressiveness of constraint logic programming languages might be increased in an analogous way, by adding some constructs that better accommodate the numerical-valued variables encountered in algebraic formulations. Here the implementational obstacles may be less severe, because logic programming system design is already geared toward adding features to deal with constraints, while the tightly integrated solvers are readily enhanced in parallel with the language-processing part of the system. Also for the solvers, there is still much to be gained by looking at the relationship between various logic programming search strategies and established branch-and-bound strategies, as in the work of Hooker *et al.* [31, 32].

5. Algebraic vs. network-based representations

We explore the differences between the algebraic and netform view of optimization by considering two representative classes of models: network flow optimization and network design. Then we conclude as in §4 by considering how the two kinds of systems under consideration may borrow from each other in the future.

The case of network flow models. For the many integer programming models that incorporate a network-flow constraint structure (flow balances at the network nodes together with flow bounds on the arcs), there exist natural algebraic representations. Nevertheless, the "flow in equals flow out" balance constraints are awkward to express in the customary terms of an algebraic modeling language, because the constraints are stated explicitly while the network node-arc structure remains implicit within them. People tend to think of network-flow models in the opposite way, with the node-arc structure stated explicitly and the balance constraints being implicit in that structure.

These considerations led to the addition of `node` and `arc` declarations to the AIMMS [5] and AMPL [16] modeling languages. As an example, Figure 5–1 shows how the planning constraints of Figure 1–2a can be reformulated to better emphasize the flow of labor and materials. In this case, the benefits of an explicit network description are combined with the strengths of a general-purpose algebraic language. Algebraic expressions can be used to specify model quantities such as cost and bounds, as well as to specify additional "side" constraints of the conventional kind on the arc flows.

A network-based system would work on a graphical representation of this problem, perhaps converting it to an integer programming formulation for solution. The inflexibility of such a system would be a serious drawback, however; changes that could easily be made to the algebraic description might each require a different extension to the network-based system's toolbox. Moreover, the large bipartite distribution portion of the network (over 80 warehouses in the problem that motivated this model) does not lend itself especially well to graphical display.

```
minimize cost;

# --------------------------------------

node CREWS:  cr_total_min <= net_out <= cr_total_max;

node P {fact};

node D {prd,dctr}:  net_in >= 0;

node W {p in prd, w in whse}:  net_in = dem[p,w];

# --------------------------------------

arc Work {f in fact} integer >= cr_min[f], <= cr_max[f],
      from CREWS,  to P[f];

arc Manu {p in prd, f in fact: pc[p,f] <> 0} >= 0,
      from P[f],  to D[p,f] (dp[f]*hd[f]/(pt[p,f]*cpp[p]/1000)),
      obj cost (pc[p,f]*dp[f]*hd[f]/pt[p,f]);

arc Ship {p in prd, (d,w) in ship_rt} >= 0,
      from D[p,d],  to W[p,w]  obj cost (sc[d,w]*wt[p]*cpp[p]/1000);
```

Figure 5–1. *The shipment constraints of Figure 1–2a, reformulated in terms of ore natural* `node` *and* `arc` *declarations in the AMPL language.*

The case of network design models. Other important kinds of network problems give algebraic modeling languages considerably more trouble. As an example we consider here the variety of network design problems, in which a network of some kind is to be chosen in the best possible way from a given collection of nodes and arcs.

Given costs on the arcs, the minimum spanning tree problem looks for a min-cost subset of arcs that form a tree (that is, contain no cycles) and that meet all nodes. There exist fast algorithms for this problem that fit well into the toolbox of a network-based optimization system. An algebraic modeling language may also be used, as shown by Figure 5–2, to express an equivalent linear program — in the sense that any optimal solution to the algebraic formulation can be interpreted as a minimum spanning tree on a corresponding network. There is nothing in Figure 5–2 that refers directly to spanning trees, however, and indeed a large degree of inspiration was needed construct an equivalent LP; even when one is told that this LP finds the minimum spanning tree, it is not so easy to come up with a proof of the fact. In sum, the algebraic language is unable to express the problem in the way that one would naturally conceive it, and is consequently a poor choice for representing such a problem.

The similar Steiner tree problem asks for the least cost tree spanning at least a designated subset of the nodes. Like the majority of network design problems that are of practical interest, this one is NP-hard; the network-based modeling system

```
set NODES := 0 .. N;
set EDGES within NODES cross NODES;

set ARCS within NODES cross NODES :=
   {i in NODES, j in NODES: (i,j) in EDGES or (j,i) in EDGES};

param weight {EDGES};
param w {(i,j) in ARCS} :=
   if (i,j) in EDGES then weight[i,j] else weight[j,i];

var Select {ARCS} >= 0, <= 1;
var Flow {ARCS,1..N} >= 0, <= 1;

minimize total_weight:
   sum {(i,j) in ARCS} w[i,j] * Select[i,j];

subject to balance_v_flow {v in 1..N, i in 1..N}:
   sum {(j,i) in ARCS} Flow[j,i,v] - sum {(i,j) in ARCS} Flow[i,j,v]
      = if (i = v) then 1 else 0;

subject to select_if_flow {(i,j) in ARCS, v in 1..N}:
   Flow[i,j,v] <= Select[i,j];

subject to sp_tree: sum {(i,j) in ARCS} Select[i,j] = N;
```

Figure 5–2. *A linear program for the minimum spanning tree problem. In the optimal solution, the spanning tree is given by those variables* `Select[i,j]` *that equal 1.*

might well offer several fast heuristics as well as slower exact algorithms for it, as discussed previously. An equivalent algebraic formulation for this problem does exist, and such integer programs have sometimes proved to be of use, but mainly as a starting point for branch-and-bound methods highly specialized to particular hard problems (notably the traveling salesman problem [3]).

Future interrelationships. Building on the success of the network node and arc declarations in algebraic modeling languages, we can imagine adding other constructs that would help these languages represent network problems more naturally. The network design example suggests, for instance, that a facility to explicitly declare trees would be useful. Although current languages can define a set of nodes, and a set of arcs represented by ordered pairs of nodes (as seen in Figure 5–2), they have no clear and concise way of specifying that a given set of arcs constitute a tree, or of referring to tree nodes through standard characteristics such as depth, successor and root. An extension that would serve this purpose has been proposed by Bisschop and Kuip [7], who describe a variety of options for declaring and using "hierarchical" sets.

Network design involves more than representation of trees in the data, however. Given a representation of a network, we want to be able to optimize over all trees spanning that network. Algebraic modeling language extensions for optimization of this general kind have been studied by Bisschop and Fourer [6], who show how various kinds of combinatorial optimization problems — and especially network optimization — can be represented as the assignment of a subset or subsequence to a *decision set* in contrast to the usual assignment of numerical values to decision variables.

Algebraic and network representations are fundamentally distinct, however, and so are ultimately more likely to complement each other than to adopt each other's features. Jones and D'Souza [39] describe how a network-based system for minimum-cost flow problems could be extended to interface with an algebraic modeling language system. MIMI/G [10] represents another possibility; the user can define in great detail the appearance of network diagrams for a model's data and results, and the resulting display can be "live" in that changing the network on the screen will change the values in the tables. (MIMI uses a block-schematic representation of linear programs, but MIMI/G's approach might be employed just as well by any algebraic modeling language system that supports tables of data and results.) Finally, a more general possibility, explored by several investigators [4, 27, 42], is for the algebraic problem statement and the network diagram to become just two of many *views* of a model between which the user can switch as desired.

6. Logic-based vs. network-based systems

Although these systems address some of the same combinatorial problems, they tend to take opposing approaches toward providing optimization methods to the modeler. Each might benefit by considering an approach that somewhat more like the other's.

Reflecting the general state of network optimization research cited previously, network-based systems have tended to rely on a toolbox of narrowly targeted methods. While continuing to expand their toolboxes, designers of network-based optimization systems may benefit by also including some of the more general-purpose search methods. There has been in particular a great deal of interest in genetic algorithms [46], simulated annealing [62] and tabu search [23, 24], all of which are general-purpose heuristic approaches to solving hard combinatorial problems. To provide for the application of these heuristics, however, a network-based optimization system would need to provide the user with a way of defining populations,

```
%%% Define set of edges "leaving" node set

out([V],Es) :- node(V), incident(V,Es).

out([V|Vs],Es) :-
  node(V),
  not member(V,Vs),
  out(Vs,Fs),
  incident(V,Gs),
  sym_dif(Fs,Gs,Es).

out([V|Vs],Es) :-
  node(V),
  member(V,Vs),
  out(Vs,Es).

%%% Minimum-weight tree generated by a list (set) of nodes

min_tree([],[V]) :- node(V).

min_tree([E|Es],[V|Vs]) :-
  min_tree(Es,Vs),
  out(Vs,Fs),
  min_member(E,Fs),
  end(E,V),
  not member(V,Vs).

%%% Minimum-weight edge of a list of edges

min_member(E,[E]) :- edge(E).

min_member(G,[F|Fs]) :-
  min_member(E,Fs),
  weight(E,We),
  weight(F,Wf),
  member(G,[E,F]),
  weight(G,Wg),
  Wg =< Wf,
  Wg =< We.

%%% Symmetric difference of 2 lists

sym_dif(Xs,Ys,Zs) :-
  subtract(Xs,Ys,Ws),
  subtract(Ys,Xs,Us),
  union(Ws,Us,Zs).
```

Figure 6–1. *Excerpts from a logic program for generating a minimum spanning tree. The query* `min_tree(Es,[X1,X2,v])` *gets the Prim's-method tree rooted at* `v`.

neighborhoods and other essential but problem-specific information. The design would then become more like that of constraint logic programming systems, in that the system provides certain solution methods as built-in features, while relying on the user to provide specific information that these methods need in order to operate efficiently on specific problems.

Reflecting their origins in logic programming and artificial intelligence, logic-based optimization systems tend to adopt heuristic search approaches that are very widely applicable. A logic program in a Prolog-based language can mimic the actions of Prim's algorithm in finding a minimum spanning tree, for example, as shown in Figure 6–1. All of the knowledge of network properties in this example has been provided by the writer of the logic program, however, and the same would have to be true in attacking more complex problems. The Prolog interpreter could not benefit from the fact that there are very fast methods for maximum flow or minimum spanning tree, which might be used in determining good bounds or feasible solutions as part of a search strategy. Just as network-based systems may need to become less problem-specific, logic-based systems may have to take a more problem-specific view to improve their effectiveness within an area such as network optimization.

7. Concluding remarks

In the spirit of this paper, we conclude by describing one hypothetical way in which all three kinds of systems considered in this paper might contribute ideas toward a more effective system for combinatorial optimization. To make the discussion concrete we use as an example the budgeted traveling salesman problem, in which we want to find the largest possible tour through some set of cities, beginning at a specified base city, subject to a limit on the overall cost of the tour.

We start from our previously mentioned idea proposed in [6] for extending algebraic modeling languages, wherein the choice of values for decision variables is supplanted by the choice of members in a decision set. Using this device, we can give a clear and concise algebraic description of the problem:

```
var_set Tour circular within CITIES;

maximize Cities_Visited: card {Tour};

subject to Budget_Limit:
   sum {c in Tour} cost[c,next(c)] <= budget;

subject to Leave_Home: first(Tour) = Home;

subject to Link_Exists {c in Tour}: (c,next(c)) in LINKS;
```

In the spirit of constraint logic programming, we endow the system that reads this model with a built-in search procedure, based on enumeration of all possible subsets, that is guaranteed in principle to find an optimal subset in a finite amount of time.

The work of explicit enumeration being far too great for any but the smallest sets, we further adopt an approach from constraint logic programming by inviting the modeler to help guide the enumeration. Specifically, we extend the algebraic modeling language to allow directives for ordering and shortening the search, as in the following examples suggested by [6] for the budgeted traveling salesman problem:

```
fathom at child node:
   (last(SubTour),Home) notin Links or
      path_cost + cost[last(SubTour),Home] > budget) and
   (path_cost + in_cost + out_cost > budget);

prune at child node:
   {c in Free: (last(SubTour),c) notin Links};

bound at child node: count {SubTour} + 2 + max_insert;

select parent node: lowest min {c in Fre} cost[last(SubTour),c];
select child node: arg min {c in Free} cost[last(SubTour),c];
```

Expressions within the directives can refer to problem-specific data and to parameters that reflect the current state of the search (some of which updated by statements now shown here). Directives such as `select` try to guide the search toward finding better solutions sooner, while `bound`, `prune` and `fathom` help the enumeration to avoid parts of the search tree where no better solutions can be found.

Finally, we observe that the above directives incorporate no real knowledge about the properties of a network of cities. Borrowing from netform-based systems, we introduce easily computed constructs such as minimum spanning trees and shortest paths, which can be used to derive tighter bounds and stronger fathoming criteria. We also introduce a graphical display of the search, which can be stopped or redirected at any point in the search. The same display naturally provides a picture of the best tour found.

References

[1] A. Aggoun, D. Chan, ... and D.H. de Villeneuve, *ECL^{i}PS^e 3.5: ECRC Common Logic Programming System: User Manual.* European Computer-Industry Research Centre (1995).

[2] R.K. Ahuja, T.L. Magnanti and J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ (1993).

[3] D. Applegate, R. Bixby, V. Chvátal and W. Cook, Finding Cuts in the TSP (A Preliminary Report). Technical Report 95-05, DIMACS, Rutgers, NJ (1995).

[4] D. Baldwin, The Development and Architecture of DOVE: A Multiview Viewpoint. *Proceedings of ISDSS Conference,* Austin, TX (1990).

[5] J.J. Bisschop and R. Entriken, *AIMMS: The Modeling System.* Paragon Decision Technology, Haarlem, The Netherlands (1993).

[6] J.J. Bisschop and R. Fourer, New Constructs for the Description of Combinatorial Optimization Problems in Algebraic Modeling Languages. Memorandum No. 901, Faculty of Applied Mathematics, University of Twente, The Netherlands (1990, revised 1994); forthcoming in *Computational Optimization and Applications.*

[7] J.J. Bisschop and C.A.C. Kuip, Hierarchical Sets in Mathematical Programming Modeling Languages. *Computational Optimization and Applications* **1**:4 (1992).

[8] J.J. Bisschop and A. Meeraus, On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study* **20** (1982) 1–29.

[9] A. Brooke, D. Kendrick and A. Meeraus, *GAMS: A User's Guide, Release 2.25.* Boyd & Fraser/The Scientific Press, Danvers, MA (1992).

[10] Chesapeake Decision Sciences, *MIMI/LP User's Manual.* New Providence, NJ (1992).

[11] A. Colmerauer, An Introduction to Prolog III. *Communications of the ACM* **33** (1990) 69–90.

[12] W. Cook, W. Cunningham, W. Pulleyblank and A. Schrijver, *Combinatorial Optimization.* Unpublished manuscript (1994).

[13] CPLEX Optimization, Inc., *Using the CPLEX Callable Library,* version 3.0. Incline Village, NV (1994).

[14] M. Dincbas, H. Simonis and P. Van Hentenryck, Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming* **8** (1990) 75–93.

[15] R. Fourer, Modeling Languages versus Matrix Generators for Linear Programming. *ACM Transactions on Mathematical Software* **9** (1983) 143–183.

[16] R. Fourer and D.M. Gay, Expressing Special Structures in an Algebraic Modeling Language for Mathematical Programming. *ORSA Journal on Computing* **7** (1995) 166–190.

[17] R. Fourer, D.M. Gay and B.W. Kernighan, A Modeling Language for Mathematical Programming. *Management Science* **36** (1990) 519–554.

[18] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming.* Boyd & Fraser/The Scientific Press, Danvers, MA (1992).

[19] A.M. Geoffrion, An Introduction to Structured Modeling. *Management Science* **33** (1987) 547–588.

[20] A.M. Geoffrion, The Formal Aspects of Structured Modeling. *Operations Research* **37** (1989) 3–15.

[21] A.M. Geoffrion, The SML Language for Structured Modeling: Levels 1 and 2. *Operations Research* **40** (1992) 38–57.

[22] A.M. Geoffrion, The SML Language for Structured Modeling: Levels 3 and 4. *Operations Research* **40** (1992) 58–75.

[23] F. Glover, Tabu Search – Part I. *ORSA Journal on Computing* **2** (1990) 4–32.

[24] F. Glover, Tabu Search – Part II. *ORSA Journal on Computing* **1** (1989) 190–206.

[25] F. Glover, D. Klingman and N.V. Phillips, Netform Modeling and Applications. *Interfaces* **20**:4 (1990) 7–27.

[26] F. Glover, D. Klingman and N.V. Phillips, *Network Models in Optimization and Their Applications in Practice.* John Wiley & Sons, New York (1992).

[27] H.J. Greenberg and F.H. Murphy, Views of Mathematical Programming Models and their Instances. Technical Report, Mathematics Department, University of Colorado at Denver, CO (1992); forthcoming in *Decision Support Systems.*

[28] P. Hansen and J. Hugé, Implicit Treatment of "Zero or Range" Constraints in a Model for Minimum Cost Foundry Alloys. *Management Science* **35** (1989) 367–371.

[29] Haverly Systems, Inc., OMNI Linear Programming System: User and Operating Manual, 1st ed. Denville, NJ (1976).

[30] D.S. Hirshfeld, Scenario Management, with Illustrations Using MathPro. IMPS Roundtable Discussion #14: Integration of Modeling, Optimization and Analysis, University of Colorado, Denver.

[31] J.N. Hooker, Logic-Based Methods for Optimization. *Operations Research Society of America Computer Science Technical Section Newsletter* **15:**2 (1994) 4–11.

[32] J.N. Hooker, H. Yan, I.E. Grossmann and R. Raman, Logic Cuts for Processing Networks with Fixed Costs. *Computers and Operations Research* **21** (1994) 265–279.

[33] M.S. Hung, W.O. Rom and A.D. Waren, *Optimization with IBM OSL.* Boyd & Fraser Publishing Company, Danvers, MA (1994).

[34] J. Jaffar and S. Michaylov, Methodology and Implementation of a CLP System. *Proceedings of the Fourth International Conference on Logic Programming,* Melbourne, Australia (1987) 196–218.

[35] C.V. Jones, An Introduction to Graph-Based Modeling Systems, Part I: Overview. *ORSA Journal on Computing* **2** (1990) 136–151.

[36] C.V. Jones, An Introduction to Graph-Based Modeling Systems, Part II: Graph Grammars and the Implementation. *ORSA Journal on Computing* **3** (1991) 180–206.

[37] C.V. Jones, Attributed Graphs, Graph-Grammars and Structured Modeling. *Annals of Operations Research* **38** (1992) 281–324.

[38] C.V. Jones, An Integrated Modeling Environment Based on Attributed Graphs and Graph-Grammars. *Decision Support Systems* **10** (1993) 255–275.

[39] C.V. Jones and K. D'Souza, Graph-Grammars for Minimum Cost Network Flow Modeling. Technical Report, Faculty of Business Administration, Simon Fraser University, Burnaby, BC (1992).

[40] C. Jordan and A. Drexl, A Comparison of Constraint and Mixed-Integer Programming Solvers for Batch Sequencing with Sequence-Dependent Setups. *ORSA Journal on Computing* **7** (1995) 160–165.

[41] G. Kahan, 1982. Walking through a Columnar Approach to Linear Programming of a Business. *Interfaces* **12:**3, 32–39.

[42] D.A. Kendrick, Parallel Model Representations. *Expert Systems With Applications* **1** (1990) 383–389.

[43] D.A. Kendrick, A Graphical Interface for Production and Transportation System Modeling: PTS. *Computer Science in Economics and Management* **4** (1991) 229–236.

[44] B. Kristjansson, *MPL Modelling System User Manual,* Version 2.8. Maximal Software Inc., Arlington, VA (1993).

[45] J.-L. Lauriere, A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence* **10** (1978) 29–127.

[46] G.E. Liepins and M.R. Hilliard, Genetic Algorithms: Foundations and Applications. *Annals of Operations Research* **21** (1989) 31–57.

[47] K. McAloon and C. Tretkoff, 2LP: Linear Programming and Logic Programming. In V. Saraswat and P. Van Hentenryck, eds., *Principles and Practice of Constraint Programming,* The MIT Press, Cambridge, MA (1995) 99–114.

[48] R.D. McBride, NETSYS — A Generalized Network Modeling System. Technical Report, University of Southern California, Los Angeles, CA (1988).

[49] S.S. Nielsen, A C++ Class Library for Mathematical Programming. In S.G. Nash and A. Sofer, *The Impact of Emerging Technologies on Computer Science and Operations Research,* Kluwer Academic Publishers, Boston (1995) 221–243.

[50] W. Ogryczak, K. Studziński and K. Zorychta, DINAS: A Computer-Assisted Analysis System for Multiobjective Transshipment Problems with Facility Location. *Computers and Operations Research* **19** (1992) 637–647.

[51] W. Ogryczak, K. Studziński and K. Zorychta, EDINET — A Network Editor for Transshipment Problems with Facility Location. In O. Balci, R. Sharda and S.A. Zenios, eds., *Computer Science and Operations Research: New Developments in their Interfaces,* Pergamon Press, New York (1992) 197–212.

[52] J.P. Paul, LINGO/PC: Modeling Language for Linear and Integer Programming. *OR/MS Today* **16:**2 (1988) 19–22.

[53] J.-F. Puget, A C++ Implementation of CLP. *Proceedings of SPICIS 94: The Second Singapore International Conference on Intelligent Systems* (1994).

[54] C. Schulte, G. Smolka and J. Würtz, Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming,* Orcas Island, Washington, USA. *Lecture Notes in Computer Science* **874** (Springer-Verlag, 1994) 134–150.

[55] R.V. Simons, Mathematical Programming Modeling Using MGG. *IMA Journal of Mathematics in Management* **1** (1987) 267–276.

[56] D. Steiger, R. Sharda and B. Leclaire, Functional Description of a Graph-Based Interface for Network Modeling (GIN). In O. Balci, R. Sharda and S.A. Zenios, eds., *Computer Science and Operations Research: New Developments in their Interfaces,* Pergamon Press, New York (1992) 213–229.

[57] D. Steiger, R. Sharda and B. Leclaire, Graphical Interfaces for Network Modeling: A Model Management System Perspective. *ORSA Journal on Computing* **5** (1993) 275–291.

[58] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques,* 2nd ed. MIT Press, Cambridge, MA (1994).

[59] U.H. Suhl and R. Szymanski, Supernode Processing of Mixed-Integer Models. *Computational Optimization and Applications* **3** (1994) 317–331.

[60] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA (1989).

[61] P. Van Hentenryck, A Logic Language for Combinatorial Optimization. *Annals of Operations Research* **21** (1989) 247–273.

[62] P.J.M. van Laarhoven and E.H.L. Aarts, *Simulated Annealing: Theory and Applications.* D. Reidel, Norwell, MA (1987).

[63] J.S. Welch, PAM—A Practitioner's Approach to Modeling. *Management Science* **33** (1987) 610–625.